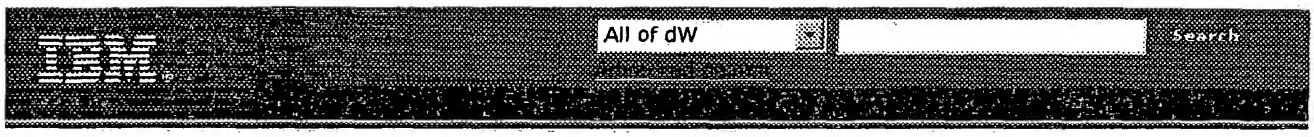


AL

Special delivery!

Bulk e-mailing with DB2, Java, JDBC, and the JavaMail API

Kulvir Singh Bhogal (kbhogal@us.ibm.com)
Java Services Consultant, IBM Corporation
February 2002

This article explains how to build a small yet powerful Java application that acts as a bulk e-mailer, and includes a complete example of using the JavaMail API. By using Java Database Connectivity (JDBC) to pull a set of e-mail addresses from DB2 into a Java program, you can e-mail messages to a target audience using the JavaMail API. The progress of the e-mailing endeavor is tracked by tapping into some of the offerings of Java Swing.

Introduction

If you have a lot of e-mail addresses in your customer database and need to deliver customized messages to each customer, this article is for you. This article shows how to use a DB2 database as a source of e-mail addresses. Accessing the database with Java is facilitated through JDBC. The actual mailing of e-mail messages is carried out with the help of the JavaMail API provided by Sun. Some Java Swing is incorporated to make things more aesthetically pleasing.

You can download a zipped file that contains all of the program code discussed in this article. I assume throughout this article that you have downloaded and looked at the accompanying code.

Gathering what you need

First, gather the pieces you need to put things together. To use Java Swing for the GUI interface, you need to have a JDK that supports Swing. Visual Age for Java Version 3.5 or 4.0 uses a JDK that has inherent Swing capability. Swing has been fully incorporated into Java 2, so using JDK Version 1.2 or higher from Sun Microsystems is OK. If you are using an older JDK, you can add Swing by simply downloading the 1.1 compatible version of the Java Foundation Classes (JFC) library. You might need to make slight modifications to the code that accompanies this article to account for legacy issues with an older JDK.

Assuming you have Java 2, all you need to do to prepare for our graphical exploits is perform the import: `import javax.swing.*;`. In this project, I do some work with event listeners to recognize some user interactivity that occurs. Accordingly, you need to perform the import: `import java.awt.event.*;` to handle these events. Interface layout is taken care of with `import java.awt.*;`. To interact with the database, import the Java SQL package with: `import java.sql.*;` Java SQL package. These packages are most likely part of your standard JDK installation if you have Java 2 installed.

You will also need a couple of packages to facilitate actually e-mailing messages. This seemingly complex task is made

easy with a freely-available API from Sun. Download a couple of zip files from this page on Sun's site. This Web page (put together by jGuru) does an excellent job of showing the fundamentals of the JavaMail API. For this project I used the JavaMail API Version 1.2. As you can se in the code, the JavaMail API works closely in tandem with the JavaBeans Activation Framework.

Don't be alarmed by the number of files you see extracted from the zip files. All you're really interested in for this project are the jar files: `activation.jar` and `mail.jar`. Add these files to your classpath system variable. Of course, the location of the files will depend on where you unzipped the files you downloaded. If you are using the VisualAge for Java IDE, it is important to note that the classpath of the IDE may be different than that of your overall system environment. Consequently, you should take file location into consideration when building the application in the IDE and deploying it as a standalone application.

Once you've added the two jar files to your classpath, you will be able to perform the import: `import javax.mail.*;` and the import: `import javax.mail.internet.*` . Doing so will provide you with the API you need to do this project.

With the JavaMail API you can build a relatively simple program like the one in this article. You could even create your own mail client program with functions similar to those in Lotus Notes, Microsoft Outlook, Netscape Messenger, or Qualcomm Eudora. In this article I merely scratch the surface of the JavaMail API, which lets you use various mail protocols, including IMAP, POP3, and SMTP. To learn more about what these protocols do, see the explanatory overview provided by Sun (see Resources for a link).

The example code taps into the SMTP protocol, which facilitates sending mail messages. You need to know your SMTP server name to make sure things run smoothly. Consult your Internet Service Provider or network administrator to find out, if necessary. A common form for SMTP server names is `smtp.yourmailservername.com`. You will need this information later.

You need to create and populate a database that will serve as the source of e-mail addresses. The example environment was IBM DB2 Version 7.2. You can use other relational databases, though IBM DB2 is well known for handling transactions and storage more quickly and efficiently than other industry-leading relational databases. At installation, DB2 should install a zip file called `db2java.zip` that you need to add to your classpath. If you kept the installation defaults of DB2, the zip file would probably be located at `C:\Program Files\SQLLIB\java\db2java.zip`.

Readying the database

First, we have to create a database with a table in it. With the IBM DB2 command line processor, do the following:

```
create database EMAILDB
connect to EMAILDB user db2admin using password
create table EMAIL_TBL (Name Char(30), Email Char(30))
```

With the above statements you create a rather simple database that contains a table with only a couple of columns. My user name, which was authorized to create the database, was *db2admin* with a password of *password*. You could have more data included for each entry, but to keep things simple for this article, I'll keep the database simple.

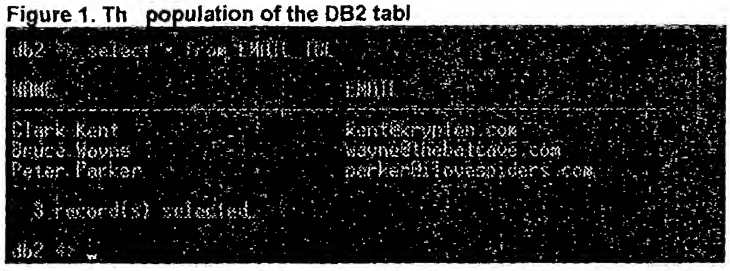
Next, you need to populate the database with a series of SQL insert statements, such as the following:

```
insert into EMAIL_TBL values ('Clark Kent','kent@krypton.com')
insert into EMAIL_TBL values ('Bruce Wayne','wayne@thebatcave.com')
insert into EMAIL_TBL values ('Peter Parker','parker@ilovespiders.com')
```

The insert statements above populate the database with names associated with corresponding e-mail addresses. Obviously, when using a bulk e-mail application you would likely be targeting an audience of more than three individuals and would have more insert statements.

The population process could be done more efficiently with stored procedures or prepared statements, but for the purposes of this article, I'm keeping it simple. Doing a `select * from EMAIL_TBL` query after doing the aforementioned inserts would give the following in the command-line processor window.

Figure 1. Th population of the DB2 tabl



Reading from the database with JDBC

The following code example is the main method, which doesn't do much itself but makes calls to get things accomplished.

The main method

```
public static void main(String[] args)
{
        accessDB();
        Emailer frame = new Emailer();
        WindowListener listener = new WindowAdapter()
        {
                public void windowClosing(WindowEvent e)
                {
                        System.exit(0);
                }
        };
        frame.addWindowListener(listener);
        frame.pack();
        frame.setVisible(true);
}
```

In the main method above, the first thing to do is read from the database. You can do this with a separate static method called `accessDB()`. The code for the `accessDB()` method is shown below.

The accessDB() method

```
public static void accessDB()
{
  try
  {
        Class.forName("COM.ibm.db2.jdbc.app.DB2Driver");
        Connection db2Conn =
                DriverManager.getConnection
                ("jdbc:db2:emailDB","db2admin","password");
        Statement st = db2Conn.createStatement();
        String myQuery = "SELECT COUNT(*) FROM EMAIL_TBL;";
        ResultSet rec = st.executeQuery(myQuery);
        rec.next();
        NUMBEROFENTRIES = Integer.parseInt(rec.getString(1));
        myQuery = "SELECT * FROM EMAIL_TBL;";
        rec = st.executeQuery(myQuery);
        while (rec.next())
        {
                String addName = rec.getString(1);
                String addEmail = rec.getString(2);
                namesVector.addElement(addName);
                emailVector.addElement(addEmail);
        }
        db2Conn.close();
  }
  catch (Exception e)
  {
      System.err.println("A database problem has" +
                " occurred - " + e.toString());
  }
}
```

By virtue of their API, many of the JDBC statements that access the database with the above method must be protected by a try block, because they might throw an exception. You should load the correct driver using the `forName(String)` method. Class, part of the java.lang package, is used to load classes into the Java interpreter with `Class.forName("COM.ibm.db2.jdbc.app.DB2Driver");`

Next you need to establish a database connection with the following syntax.

```
Connection db2Conn =  DriverManager.getConnection
        ("jdbc:db2:emailDB","db2admin","password");
```

Notice the inclusion of the database name (emailDB) you created earlier, and the user ID and password that are needed to successfully connect to the database.

SQL statements are represented in Java by Statement objects. Note that Statement is an interface that cannot be instantiated directly. It is returned by the `createStatement()` method of a connection object with `Statement st=db2Conn.createStatement();`.

You can use this Statement object to conduct SQL queries by calling the `executeQuery` method of the object with these statements:

```
String myQuery = "SELECT COUNT(*) FROM EMAIL_TBL;";
ResultSet rec = st.executeQuery(myQuery);
```

Execution of a query returns a ResultSet object. I used the `getString` method to retrieve contents from the ResultSet object with `NUMBEROFENTRIES = Integer.parseInt(rec.getString(1));`. Here you store the number of entries in the database (acquired with the SQL query above) into a static variable named NUMBEROFENTRIES. You'll use this static variable later on.

Next, grab all of the data from the database and cycle through each record set, populating a couple of vectors with what is grabbed. Of course, it would be more elegant to use a vector of objects, where objects could consist of the String properties of e-mail and name. But, for ease of explanation, I used a pair of parallel vectors to store what I grabbed from the database. If the database were more complex, a more object-oriented methodology would be the "proper" approach:

Populating vectors

```
while (rec.next())
{
      String addName = rec.getString(1);
      String addEmail = rec.getString(2);
      namesVector.addElement(addName);
      emailVector.addElement(addEmail);
}
```

To wrap things up with the `accessDB()` method, close the database connection with the syntax `db2Conn.close();`.

Tracking progress with Swing

Now, return to the main method. After doing the database work, call the Emailer constructor to take care of setting up the GUI, as follows.

The Emailer() constructor

```
public Emailer()
{
        super("Mailing List Progress");
        JPanel top = new JPanel();
        top.setLayout(new FlowLayout());
        myProgressBar = new JProgressBar(0,NUMBEROFENTRIES);
        myProgressBar.setValue(0);
        myProgressBar.setStringPainted(true);
        top.add(myProgressBar);
        myTextArea = new JTextArea(8,30);
        myTextArea.setLineWrap(true);
        JScrollPane scroller = new JScrollPane(myTextArea);
        JPanel bottom = new JPanel();
        bottom.setLayout(new FlowLayout());
        sendButton = new JButton("Send mail to mailing list.");
        sendButton.addActionListener(this);
        bottom.add(sendButton);
        JPanel pane = new JPanel();
        pane.setLayout(new BorderLayout());
        pane.add("North", top);
        pane.add("Center", scroller);
        pane.add("South", bottom);
        setContentPane(pane);
}
```

The code above has some simple Swing syntax. Because the user interface is not the essence of the project, I'll discuss it only briefly. Note that the core functions of the coding experiment could be done with no user interface. However, *presentation is everything, so I provide a snazzy way to interact with the program.* To get more insight into what is being done, consult the chapter on Swing interface design in your nearest Java 2 book.

I set up three JPanels consisting of:

- **A progress bar** to track the progress of the bulk e-mailing task. The progress bar is placed in its own *FlowLayout*. The lines:
  `myProgressBar = new JProgressBar(0,NUMBEROFENTRIES);`
  `ProgressBar.setValue(0);`
  set up the operating range for the progress bar, starting from 0 to NUMBEROFENTRIES. Remember, NUMBEROFENTRIES is a static integer to which you assigned a value in your `accessDB()` method. The integer was given the value of the number of e-mail entries in the database (as extrapolated with a simple SQL query). Before things begin, I set the current value of the progress bar to zero.
- **A text area** to let the user know what current database item is being worked on and how successful the program has been in sending the e-mail to the e-mail address entry being processed. Since you'll probably be dealing with a number of e-mail addresses, I put the text area in a JScrollPane.
- **A button** on the interface to start the actual e-mailing task. This GUI function is provided with a JButton that I associated with an ActionListener. The ActionListener "listens" for the action of pressing the JButton.

After establishing the JPanels, add them into a new panel that uses the BorderLayout layout design.

Next, return to the main method, where you'll add a WindowListener that allows the user to close the program's GUI and consequently close the application, as follows:

```
WindowListener listener = new WindowAdapter()
{
        public void windowClosing(WindowEvent e)
        {
                System.exit(0);
        }
};
frame.addWindowListener(listener);
```

With this setup, when you press the close button on the GUI, the `System.exit(0)` method is called, which forces the program to end.

Next, with the code

```
frame.pack();
frame.setVisible(true);
```

you pack the frame, which shrinks the frame to the smallest possible size, while still allowing it to contain all of the components. The frame is then made visible so you can see the fruits of your labor.

When the GUI button labeled "Send to mailing list" is pressed, the following method handles the event.

```
public void actionPerformed(ActionEvent userInteractiveEvent)
{
        Object sourceOfEvent = userInteractiveEvent.getSource();
        if (sourceOfEvent == sendButton)
        {
            Thread runner = new Thread(this);
            runner.start();
        }
}
```

With this method, the program waits for an action to be performed by the user. Since the user interactivity is limited to clicking **Send mail to mailing list**, handling the single event is not that difficult. In the `actionPerformed` method, first the source object of the event is analyzed. If the source of the action is the Jbutton, then you create a new thread. The `start()` method of the thread is then called. As the science of Java threads would have it, the `start()` method calls the `run()` method behind the scenes. Hence, it is in the `run()` method that you handle the GUI updates and mail addressing setup, as follows.

The run() method

```
public void run()
{
        sendButton.setEnabled(false);
        int currentEntry = 0;
        myTextArea.setText("");
        while(currentEntry < NUMBEROFENTRIES)
        {
                String sendToEmail =
                        (String)emailVector.elementAt(currentEntry);
                String sendToName =
                        (String)namesVector.elementAt(currentEntry);
                myTextArea.append("Sending Message to: " +
                        sendToEmail + "\n");
                try
                {
                        sendMessage(sendToEmail,sendToName);
                }
                catch (Exception e)
                {
                        myTextArea.append("Problem " +
                                " sending message to " +
                                sendToName+"\n");
                        System.err.println("Exception thrown " +
                                " while sending message: ");
                        System.err.println(e.toString());
                }
                currentEntry++;
                myProgressBar.setValue(currentEntry);
        }
        myTextArea.append("\nCompleted Sending of Messages\n");
        sendButton.setEnabled(true);
}
```

The `run()` method first greys out the GUI button using the *setEnabled(false)* syntax. This is done to ensure that the user doesn't spawn multiple threads during the mailing process. If this happened, there would probably be some annoyed e-mail recipients with a deluge of e-mail. I use the variable *currentEntry* to keep track of which database entry I'm dealing with at a particular time.

With the while loop, the program cycles through the entries grabbed from the database (with the `accessDB()` method). In this loop, you grab the database entries that you housed in the parallel vectors -- emailVector and namesVector. An explicit cast to a String is required since the `elementAt` method returns an Object. I use the append method to update the JTextArea, giving the user a status prompt about the recipient I'm trying to send a message to.

The actual sending of a message is carried out by the `sendMessage` method (discussed in just a bit). The `sendMessage` method throws an exception that you are subsequently required to catch, as seen in the try, catch clause. If an exception is thrown, you let the user know there was a problem sending the message. I do not display the actual exception to the user, because the esoteric message might be too high-level for our targeted customer base. You can use the System error console to display errors for troubleshooting or diagnosis. If you have problems while using the example application, you could analyze the System error console for insight into what might be wrong.

After processing the vector entry, increment the currentEntry counter before you move on to the next database entry. The progress bar is updated by using the syntax `myProgressBar.setValue(currentEntry);`.

The `setValue` method does some behind-the-scenes work by calculating what percentage of the task has been completed. For example, if you are sending 10 messages and 2 have been processed, then the Swing API for the JProgressBar automatically calculates that 20% of the task is done. This progress is reflected in the GUI display.

After the e-mailing task is completed, you can notify the user with a completion prompt to the text area and re-enable the button so the mailing process can be repeated. Be aware that every time the button is pressed, messages will be sent out. E-mail etiquette tells us not to spam, and angry recipients inundated with spam will tell you, too.

Going to the post office

Now I'll discuss the `sendMessage` method. Don't worry... the JavaMail API is easy to use. This method accomplishes the

task of e-mailing your messages. The `sendMessage` method does the trick below.

Th sendMessag method

```
public static void sendMessage(String sendToMeEmail,
        String sendToMeName)
throws Exception
{
        String smtpServerName = "smtp.mysmtpservername.com";
        String from = "professorX@themansion.com";
        Properties myProperties = System.getProperties();
        myProperties.put("mail.smtp.host", smtpServerName);
        Session session =
                Session.getDefaultInstance(myProperties, null);
        MimeMessage message = new MimeMessage(session);
        message.setFrom(
                new InternetAddress(from, "Professor Xavier"));
        message.addRecipient(Message.RecipientType.TO,
                new InternetAddress(sendToMeEmail));
        message.setSubject("Hello JavaMail");
        String myMessageText = "Hello " + sendToMeName + ".\n";
        myMessageText+= "This message was sent out via a " +
                " bulk e-mailer developed using Java and the " +
                " JavaMail API.";
        message.setText(myMessageText);
        Transport.send(message);
}
```

The `sendMessage` method takes two parameters -- e-mail address and associated name. If you recall, these values are obtained from our parallel vectors. You now need your SMTP server name to populate the string "ServerName" with it. This string is used to establish the e-mail environment settings. Next, you need to state where the messages will come from by populating the string "from."

It is important to note that many SMTP servers require e-mail that is sent out to be from an address that is recognized by the SMTP server. For example, in some setups, if you try to send an e-mail from `somerandomeaddress@somerandomdomain.com` while your SMTP server is `smtp.someotherdomain.com`, your SMTP server might balk because it does not recognize the address as one of the authorized users. Talk to your network administrator if you have this problem when sending messages.

The example mail session is defined with the Session class, which taps into the `java.util.Properties` object to get information needed for the mailing task. Accordingly, you must provide the Properties object, myProperties, with the correct mail environment settings. Using the syntax

```
myProperties.put("mail.smtp.host", smtpServerName);
Session session = Session.getDefaultInstance(myProperties, null);
```

you inform the Session object about the SMTP server name. The null argument in the code snippet above is of an Authenticator object. I assume here that your SMTP server does not require you to authenticate yourself when you are mailing something. If authentication is required, you would need to provide the correct values for this Authenticator object parameter.

Since you have the Session object, now you can move on to building the actual message, which is of type MimeMessage (a subclass of the abstract class Message). MimeMessages lets you use non-ASCII characters, HTML, images, and other eye candy in your e-mail messages. In this example, I keep the message simple using the MIME type of text/plain:

```
message.setFrom(new InternetAddress(from, "Professor Xavier"));
message.addRecipient(Message.RecipientType.TO, new InternetAddress(sendToMeEmail));
message.setSubject("Hello JavaMail");
String myMessageText = "Hello " + sendToMeName + ".\n";
myMessageText+= "This message was sent out via a " +
" bulk e-mailer developed using Java and the " +
" JavaMail API.";
message.setText(myMessageText);
```

As shown in the code above, the first thing to do is define the sender e-mail address and sender name, then set this

property in the message object. The e-mail address is set to the String "from" and the sender name is hard coded to be "Professor Xavier." The setFrom method expects an `InternetAddress` object. Accordingly, we instantiate a new obj ct with the InternetAddress constructor.

Next, add the recipient to the e-mail message. You can send an e-mail using either the "To" field or by carbon copying or blind carbon copying. With JavaMail, the predefined types of addresses are:

```
Message.RecipientType.TO
Message.RecipientType.CC
Message.RecipientType.BCC
```

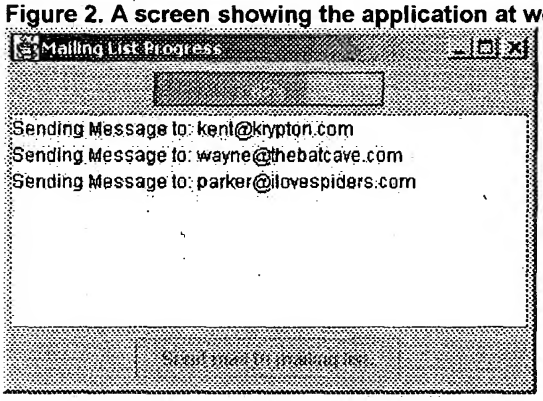
In the example code, I simply use the "To" field and added the recipient using the `addRecipient` method in the API. The subject of the message is added using the method `setSubject`. The API is rather straightforward. You can provide a body for the message by adding some text. In this case, the message says hello to the name of the person being e-mailed and informs them they are receiving the message courtesy of the bulk e-mailing system. The body of the message is added to the message using the `setText` method.

Finally, you shoot the message off to its recipient using the code `Transport.send(message);`.

The Transport class speaks the dialect of SMTP and takes care of getting the message on its way to its potential readers.

As is true with any Java code, you will need to compile your code and then run it to see the application in action. If everything goes smoothly, you should have a small Java application that packs a powerful punch. Figure 2 below shows the example application in the middle of an e-mailing job.

Figure 2. A screen showing the application at work



Conclusion

By implementing the code that accompanies this article, you have tapped into some of the powerful facilities made possible by Java. You tied together a front-end GUI that used Java Swing with a back-end DB2 database. The communication was facilitated by JDBC. Then, you used the JavaMail API to send customized e-mail to a target audience you had housed in your back-end database.

The code with this article was developed for instruction purposes. After some scrutiny, you'll see that the application is not very efficient and could be tweaked by tapping into the more advanced features provided by Java, JDBC, and JavaMail. Using the knowledge from this article, you should be able to confidently explore and use the advanced features of the Java facilities discussed to create your own powerful Java-based applications.

Coming soon

In a follow-up article, I'll tackle the challenge of how to manage an e-mailing list, including how to use Java servlets housed in WebSphere that enable users to subscribe/unsubscribe from a DB2-centric e-mail list.

Resources

- Participate in the discussion forum on this article by clicking **Discuss** at the top or bottom of the article.
- Download the zipped example source code.
- Learn more about various mail protocols in the explanatory overview provided by Sun.
- Learn more about the fundamentals of the JavaMail API on Sun's site.
- Read Kulvir's other articles: "An honest car lot on the Web" (*developerWorks*, April 2002), "User-power and e-mailing" (*developerWorks*, May 2002), and "Enforcing Business Logic using DB2 Triggers, Java UDFs, and the JavaMail API" (*DB2 Developer Domain*, May 2002).
- Find more articles and tutorials for developers on DB2 developer domain.

About th author

Kulvir Singh Bhogal works as an IBM consultant, devising and implementing Java-centric solutions at custom r sites across the nation. You can contact Kulvir at kbhogal@us.ibm.com.

Discuss ZIP PDF e-mail it

What do you think of this article?

(•) Killer! (5) ( ) Good stuff (4) ( ) So-so; not bad (3) ( ) Needs work (2) ( ) Lame! (1)

S nd us your comments or click Discuss to share your comments with others.

Submit feedback